Efficient Semantic Service Discovery in Pervasive Computing Environments

Sonia Ben Mokhtar, Anupam Kaul, Nikolaos Georgantas, and Valérie Issarny

Inria Rocquencourt
78153 Le Chesnay, France
{Sonia.Ben_Mokhtar, Anupam.Kaul,
Nikolaos.Georgantas, Valerie.Issarny}@inria.fr

Abstract. Service-oriented architectures, and notably Web Services, are becoming an incontrovertible paradigm for the development of applications in pervasive computing environments, as they enable publishing and consuming heterogeneous networked software and hardware resources. Combined with Semantic Web technologies, in particular ontologies, Web services' descriptions can be unambiguously and automatically interpreted in open pervasive computing environments, where agreement on a single common syntactic standard for identifying service semantics cannot be assumed. Nevertheless, efficient matching of semantic Web services to effectively automate the discovery and further consumption of networked resources remains an open issue, which is mainly attributable to the costly underlying semantic reasoning. After analyzing the cost of ontology-based semantic reasoning, which is at the heart of the matching process, we propose a solution towards efficient matching of semantic Web services. We have further incorporated our solution into a service discovery protocol aimed at open pervasive computing environments that integrate heterogeneous wireless network technologies (i.e., ad hoc and infrastructure-based networking). Experimental results show that our solution enables better response times than of classical syntactic-based service discovery protocols.

1 Introduction

The pervasive computing vision is increasingly enabled by the large success of wireless networks and devices. In pervasive computing environments, heterogeneous software and hardware resources may be discovered and integrated transparently towards assisting the performance of daily tasks. Still, realizing this vision requires middleware support for dynamic and automated discovery and composition of software and hardware resources that populate the pervasive computing environment. Service-oriented architectures [11], and particularly Web Services[1], have proved to be an appropriate architectural paradigm offering middleware support for pervasive computing environments. Indeed, Service-Oriented Architecture (SOA) is an architectural style that aims at the development of highly autonomous, loosely coupled systems that are able to communicate, compose and evolve in open, dynamic and heterogeneous environments such as pervasive computing environments [5]. Web Services are then one of the realizations

[1] http://www.w3.org/2002/ws/

M. van Steen and M. Henning (Eds.): Middleware 2006, LNCS 4290, pp. 240–259, 2006.
© IFIP International Federation for Information Processing 2006

of this architectural style. Using Web Services, each networked resource is abstracted as a service that is described in a declarative manner using the Web Services Description Language (WSDL) and is accessible by means of standard protocols such as the Simple Object Access Protocol (SOAP) on top of Internet protocols (HTTP, SMTP). Furthermore, Web Services have already been used in pervasive environments and have proved to be efficient when deployed on mobile, resource-constrained devices [7].

Abstracting software and hardware resources of the pervasive computing environment as Web services allows having a homogeneous vision of heterogeneous resources. Resources can then be discovered based merely on their WSDL interfaces. However, while using Web Services allows addressing substantially the heterogeneity issue in terms of technologies of service implementation, another issue remains, which is *syntactic heterogeneity*. Indeed, WSDL-based service discovery relies on the syntactic conformance of the required interfaces with the provided ones, for which common understanding is hardly achievable in open pervasive computing environments. A solution to this issue can be provided by introducing semantics into the service description. Combined with Semantic Web technologies[2], notably ontologies, for the semantic description of the services' functional and non-functional features, Web services can be automatically and unambiguously discovered and consumed in open pervasive computing environments. Specifically, ontology-based semantic reasoning enables discovering networked services whose published provided functionalities (or *capabilities*) match a required functionality, even if there is no syntactic conformance between them. A number of research efforts have been conducted in the area of semantic Web service specification, which have led to the development of various semantic service description languages, e.g., OWL-S[3], WSDL-S[4], WSMO[5], SWSO[6]. In this context, we have developed the Amigo-S service description language [2], which is specifically aimed at services in pervasive computing environments.

Building upon the features of Amigo-S that supports specifying services in rich, open pervasive computing environments, this paper focuses on associated middleware support for effectively enabling the discovery of networked Amigo-S services. Specifically, we introduce a dedicated Service Discovery Protocol (SDP) that enables advertising and discovering services in pervasive environments according to the semantics of networked services and of sought functionalities. This is to be contrasted with traditional SDPs that support the discovery of services according to syntactic interface descriptions, and thus assume worldwide knowledge and agreement about service interfaces. The key contribution of our work then comes not only from introducing an SDP for the discovery of semantic Web services in pervasive environments, but also from the fact that our SDP offers performance that makes it appropriate for use in highly dynamic networked environments populated by resource-constrained, wireless devices. The latter issue is a major challenge due to the performance and resource costs of ontology-based semantic reasoning. This has led us to introduce a solution to lightweight semantic matching

[2] http://www.w3.org/2001/sw/

[3] OWL-S: Semantic Markup for Web Services. http://www.daml.org/services/owl-s

[4] WSDL-S: http://lsdis.cs.uga.edu/projects/meteor-s/wsdl-s/

[5] WSMO: Web Services Modeling Ontology. http://www.wsmo.org/

[6] SWSO: Semantic Web Service Ontology. http://www.daml.org/services/swsf/1.0/overview/

of Web services towards the actual exploitation of semantic Web services in pervasive environments.

The next section provides an overview of semantic Web service technologies, and introduces Amigo-S and an associated matching relation for semantic Web services in pervasive environments. Based on Amigo-S, we present a solution to lightweight semantic matching of networked services (Section 3). Our solution optimizes ontology-based semantic reasoning, which is at the heart of the matching process. Furthermore, we propose a classification of service advertisements, towards efficient access and retrieval of services within cooperating service directories deployed on the network. We have further integrated our solution in the Ariadne service discovery protocol [12] extending it to S-Ariadne (i.e., Semantic-Ariadne), which is aimed at pervasive computing environments, for hybrid wireless networks combining ad hoc and infrastructure-based networking (Section 4). Experimental results show that S-Ariadne enables better response times than of classical service discovery protocols, and is further more scalable (Section 5). Finally, we summarize our contribution and sketch perspectives for our work (Section 6).

2 Semantic Web Services for Pervasive Computing

As pointed out in the previous section, semantic Web services can provide an adequate solution to effective service discovery in open pervasive computing environments. In this section, we briefly discuss base technologies supporting the provisioning of semantic Web services, and introduce basic elements of our approach for describing and matching semantic Web services in pervasive environments. We thus discuss semantic Web services (Section 2.1), the Amigo-S language for the description of pervasive services (Section 2.2), our definition of a base semantic matching relation (Section 2.3), and a study on the cost of semantic service matching (Section 2.4).

2.1 Semantic Web Services

Ontologies may conveniently be exploited to semantically model Web services. Indeed, while Web services interfaces all have a similar structure, thanks to the WSDL standard, the semantics underlying these interfaces cannot be inferred from their syntactic description. Similarly, it cannot be assumed that service providers and consumers will use worldwide the very same syntactic interface for describing the same service, as these descriptions are created by different organizations, communities and individuals all over the world. A natural evolution of Web services description has thus been the combination of the Semantic Web and Web Services paradigms towards the semantic representation of the services functional features, leading to *Semantic Web Services*. A number of research efforts have in particular been undertaken towards the concretization of this paradigm. In this area, various languages have been proposed to describe semantic Web services, e.g., WSDL-S, OWL-S, WSMO and SWSO.

Among them, OWL-S is the effort directly related to OWL[7] (the Ontology Web Language), which is a W3C recommendation. A service description in OWL-S is composed of three parts : the *service profile*, the *process model* and the *service grounding*.

[7] http://www.w3.org/TR/owl-ref/

The service profile gives a description of a service and its provider. It is generally used for service publication and discovery. The process model is a representation of the service conversation, i.e., the interaction protocol between a service and its client that is described as a process. The service grounding specifies the information that is necessary for the service invocation, such as the communication protocol, message formats and addressing information. The OWL-S service grounding is based on WSDL.

2.2 Amigo-S for Pervasive Services

OWL-S and the other languages mentioned above provide adequate solutions for the description of semantic Web services. However, these languages are primarily aimed at characterizing stationary services deployed on the core Internet and lack key features to thoroughly model services to be provisioned in the pervasive computing environment. Such features include characterizing the specifics of the underlying middleware platform that vary significantly among networked services. For example, services networked in the pervasive home environment illustrate such diversity, as they span the home automation, consumer electronics, mobile and personal computing application domains, and further require middleware-layer bridging to be interoperable [1,4]. Another key feature of pervasive services is the need for awareness of context and quality of service, as these two factors affect decisively the actual user's experience in pervasive environments that vary greatly in resource availability and contextual conditions [8,10]. Such specifics of pervasive services has led us to introduce the *Amigo-S* service description language that meets the requirements of pervasive services.

The key novel features of the Amigo-S language are that it supports heterogeneous service infrastructures and enables QoS- and context-awareness for service provisioning. Amigo-S is an ontology formally specified in OWL; it has been developed as part of the effort of the IST Amigo project[8]. The Amigo-S specification incorporates the OWL-S specification, and extends it by adding new classes and properties. In this way, we reuse established features of OWL-S and provide a new language that can easily be used by developers already familiar with OWL-S. In the following, we briefly introduce only the Amigo-S service profile; a more detailed description of Amigo-S may be found online [2]. In this paper, we mainly exploit the ability of the Amigo-s language for specifying service functional features, while other aspects of the language, such as the description of the services' underlying middleware, as well as the specification of QoS and context properties can be exploited like in [1,2].

As discussed above, the OWL-S service profile models a service as both a semantic concept by specifying the service category and a set of semantic IOPEs. In Amigo-S as well, a service is described with a service profile. However, we assume that a service may offer a number of *capabilities*, i.e., specific functionalities offered by the service, and we explicitly model such capabilities. OWL-S actually supports multiple profiles for a service; nevertheless, using a different profile for each capability of a service does not allow capabilities to share a set of common attributes, which may globally characterize the service. In Amigo-S, each such capability is defined as both a semantic concept and a set of semantic IOPEs. This enables describing richer services supporting several capabilities that may be functionally independent or even dependent. For instance a

[8] http://www.hitech-projects.com/euprojects/amigo/

complex capability may be composed of simpler capabilities, each one of which is also separately accessible. Further, we explicitly model *provided capabilities* as capabilities supported by a service, and *required capabilities* as capabilities needed by a service, which will be sought on other networked services. This enables support for any service composition scheme, such as a peer-to-peer scheme or a centrally coordinated scheme.

An example of service profiles as enabled by Amigo-S (restricted to service inputs, outputs and category) is depicted in the upper part of Figure 1. Along with service descriptions, the figure includes in its lower part two ontologies representing the concepts employed in the service descriptions. The service on the PDA *requires* a capability named GetVideoStream, which belongs to the service category VideoServer, takes as input a title of a VideoResource and provides as output an actual Stream. The service on the workstation *provides* two capabilities, SendDigitalStream and ProvideGame, which share common attributes such as the workstation resources available to them. For the former, service category is DigitalServer, input is DigitalResource and output is Stream, while for the latter, service category is GameServer, input is GameResource and output is Stream. These two capabilities are dependent, as SendDigitalStream *includes* ProvideGame, but are separately accessible. Thus, a peer service (in other words, a client) may access the former and have the option to access a video resource, a sound resource or a game; or access the latter, asking specifically for a game. The peer service on the PDA asking for a video resource should access SendDigitalStream, which also includes GetVideoStream. Making the right choice is supported by service matching, which is described in the following two sections.

2.3 Semantic Matching Relation

Based on the Amigo-S service specification, we define a matching relation, i.e., $Match(C_1, C_2)$, which allows identifying whether capability C_1 is equivalent or includes capability C_2, i.e., if C_1 can substitute C_2 in the provisioning of a service capability that is semantically characterized by C_2 (see the example of SendDigitalStream and GetVideoStream in Figure 1). The $Match$ relation then constitutes the basis of service discovery, as seeking a capability characterized by C amounts to discovering any networked service advertising a capability described by N such that $Match(N, C)$ holds. Additionally, the $Match$ relation may conveniently be exploited to group similar capabilities of networked services towards efficient service discovery, as further presented in the next section.

Specifically, the $Match$ relation is defined using the function $distance(concept_1, concept_2)$, hereafter denoted by $d(concept_1, concept_2)$, which gives the semantic distance between two concepts, $concept_1$ and $concept_2$, as given in the classified[9] ontology to which the concepts belong. Precisely, if $concept_1$ does not subsume $concept_2$ in the ontology to which they belong to, the distance between the two concepts does not have a numeric value, i.e., $d(concept_1, concept_2) = NULL$. Otherwise, i.e., if $concept_1$ subsumes $concept_2$, the distance takes as value the number of levels that separate $concept_1$ from $concept_2$ in the ontology hierarchy.

[9] Ontology classification is the result of semantic reasoning on ontology specifications. It allows inferring implicit relationships between concepts from the explicit definitions of these concepts.

Formally, let the provided capability C_1 be defined by the set of expected inputs $C_1.In$ and set of offered outputs $C_1.Out$, and the required capability C_2 be defined by the set of offered inputs $C2.In$ and the set of expected outputs $C_2.Out$. Furthermore, let the capability C_1 define a set of provided properties $C_1.P$, and the capability C_2 define a set of required properties $C_2.P$, where these properties describe all the information that can be required in the user request such as the service category and non-functional properties; currently, we only consider the former property. The relation $Match$ is then defined as:

$$\begin{aligned} Match(C_1, C_2) = & \forall in' \in C_1.In, \exists in \in C_2.In : d(in, in') \geq 0 \text{ and} \\ & \forall out' \in C_2.Out, \exists out \in C_1.Out : d(out, out') \geq 0 \text{ and} \\ & \forall p' \in C_2.P, \exists p \in C_1.P : d(p, p') \geq 0 \end{aligned}$$

From the above, the relation $Match(C_1, C_2)$ holds if and only if all the expected inputs of C_1 are matched with inputs offered by C_2, all the expected outputs of C_2 are matched with outputs offered by C_1, and all the required properties of C_2 are matched with properties provided by C_1.

Furthermore, we define the function $SemanticDistance(C_1, C_2)$, which gives the semantic distance between the capability C_1 and the capability C_2:

$$\begin{aligned} SemanticDistance(C_1, C_2) = & \sum_{i=1}^{n_1} d(C_2.In_i, C_1.In_i) + \\ & \sum_{i=1}^{n_2} d(C_1.Out_i, C_2.Out_i) + \\ & \sum_{i=1}^{n_3} d(C_1.p_i, C_2.p_i) \end{aligned}$$

where n_1 is the number of inputs expected by C_1, n_2 is the number of outputs expected by C_2, and n_3 is the number of additional properties required by C_2. The semantic distance between capabilities corresponds to the sum of the distances between the pairs of related concepts in the advertisement and the request. This allows scoring service advertisements with respect to the requested capability with which they are being compared, and selecting the advertisement whose description best fits the user's requirements. An example of matching semantic service capabilities is shown in the middle part of Figure 1. In the figure, the requested capability GetVideoStream is matched with the provided capability SendDigitalStream, using the two underlying ontologies describing digital resources and servers. The relation $Match(SendDigitalStream, GetVideoStream)$ holds, and the semantic distance between these capabilities is equal to 3.

2.4 Cost of Semantic Matching

Practically, the semantic matching of service capabilities decomposes in three tasks:

1. Parsing the description of the requested and the provided capabilities;
2. Loading and classifying the ontologies used in both the requested and the provided capabilities using a semantic reasoner; and
3. Finding subsumption relationships between inputs, outputs and properties of the requested and provided capabilities in the classified ontologies.

Implementation and evaluation of semantic matching of service capabilities has been presented in the literature, e.g., see [9]. Results show that matching semantic service capabilities is a computation-intensive task with high response times compared to classical syntactic-based service discovery protocols. In particular, results show that the most

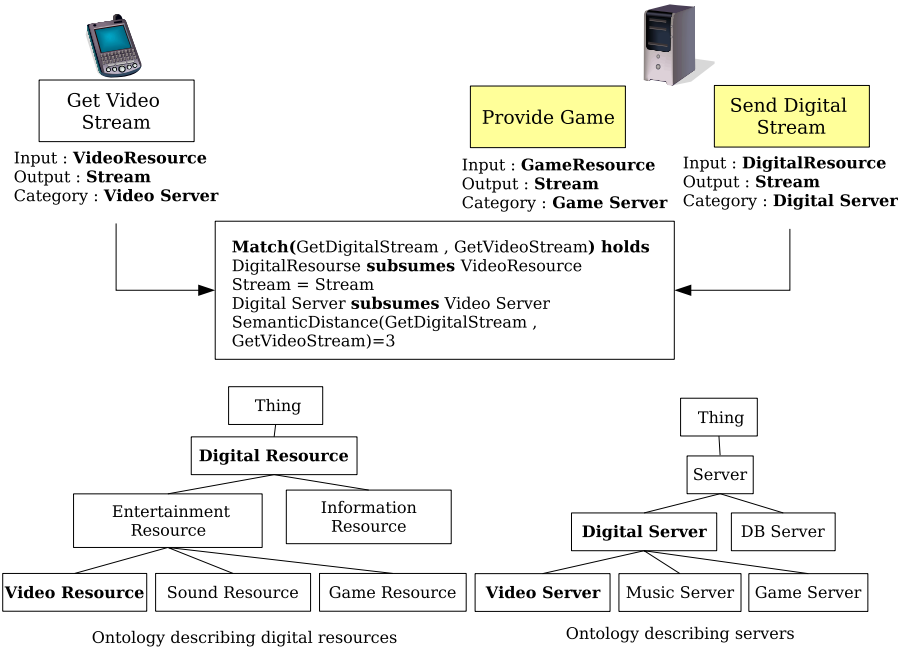

Fig. 1. Describing and matching capabilities of pervasive services

expensive phase in the process of matching semantic service capabilities is that of semantic reasoning (steps 2 and 3 above). As an illustration, Figure 2 shows an evaluation of the semantic matching of two capabilities using three different semantic reasoners: Racer[10], Fact++[11] and Pellet[12], which are the most popular semantic reasoning tools. The two capabilities have 7 inputs and 3 outputs each. The ontology used for the experiment contains 99 OWL classes, i.e., concepts, and 39 properties, i.e., relationships between the classes. We can notice that for any of the three reasoners, the average time to match two capabilities is around 4 to 5 seconds, which is much higher than classical syntactic-based matching of Web services (e.g., around 160 ms for a UDDI registry [13]). Furthermore, we notice that the time to load and classify ontologies takes from 76% to 78% of the total time for matching using any of the three reasoners.

The above results show that matching semantic Web service capabilities is an expensive task in terms of response time and resource consumption, which is not acceptable for a service discovery protocol aimed at pervasive computing environments, where service discovery needs to be efficient enough to ensure service availability despite the network's dynamics, and lightweight enough for use by thin, wireless devices. Thus, in order to enable actual deployment of semantic Web services in pervasive computing environments, a number of optimizations have to be introduced in the process of match-

[10] Racer: http://www.sts.tu-harburg.de/ r.f.moeller/racer/
[11] Fact++: http://owl.man.ac.uk/factplusplus/
[12] Pellet: http://www.mindswap.org/2003/pellet/

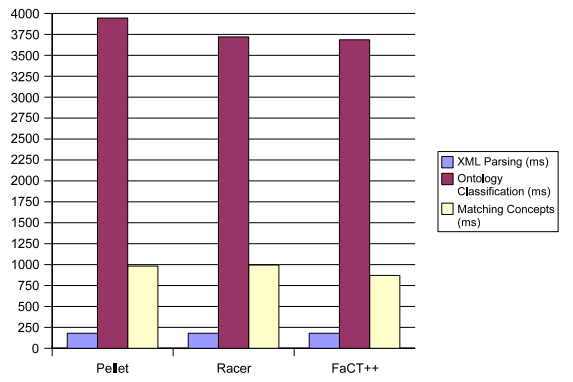

Fig. 2. Time taken to match a requested and a provided capability

ing semantic service capabilities, particularly, targeting acceptable response times. The next section introduces such solutions, building upon recent efforts in the area of efficient semantic service matching.

3 Achieving Lightweight Discovery of Semantic Web Services

Lightweight discovery of semantic Web services requires minimizing the overhead due to semantic reasoning, possibly performing it off-line so that semantic reasoners do not need to be used when advertising and seeking networked services. Specifically, optimization can be introduced at two levels. First, at the semantic reasoning level, by reducing the time spent to infer relationships between concepts in ontologies. Second, at the service discovery level, by classifying directories of services in a way that reduces the number of semantic matches performed to answer a user request. As discussed below (Section 3.1), related optimizations for both ontology-based semantic reasoning and classification of service advertisements have been proposed in the literature [3,13]. We then propose an effective solution to the discovery of semantic Web services in pervasive computing environments (Sections 3.2, 3.3).

3.1 Background

In [3], the authors emphasize the need of efficient indexes and search structures for directories. Towards this goal, they propose to numerically encode service descriptions given in OWL-S. This is done by numerically encoding ontology class and property hierarchies by intervals. More precisely, each class (resp. property) in a classified hierarchy is associated with an interval. Then, each service description maps to a graphical representation in the form of a set of rectangles defined by the sets of intervals representing properties combined with the set of intervals representing classes. Furthermore, for efficient service retrieval, the authors base their work on techniques for managing multidimensional data being developed in the database community. More precisely, they use the Generalized Search Tree (GiST) algorithm proposed by Hellerstein in [6]

for creating and maintaining the directory of numeric services. Combining both encoding and indexing techniques allows performing efficient service search, in the order of milliseconds for trees of 10000 entries. However, insertion within trees of the previous size is still a heavy process that takes approximately 3 seconds.

In [13], the authors propose an approach to optimize service discovery in a UDDI registry augmented with OWL-S for the description of semantic Web services. This approach is based on the fact that the publishing phase is not a time critical task. Therefore, the authors propose to exploit this phase to pre-compute and store information about the incoming services. More precisely, a taxonomy that represents the subsumption relationships between all the concepts in the ontologies used by services is maintained. Then, each concept C in this taxonomy is annotated with two lists, one to store information about inputs of services while the other one is used to store information about outputs of services. More precisely, for each concept in the taxonomy, these lists specify to what degree any request pointing to that concept would match the advertisement. For example, for a particular concept C in the taxonomy, the list storing information about outputs is represented as $[< Adv_1, exact >, < Adv_2, subsumes >, ...]$, where Adv_i points to a service advertisement in the repository and $exact$ (resp. $subsumes$) specify the degree of match between C and the related concept in the corresponding advertisement. A performance evaluation of this approach shows that the publishing phase using this algorithm takes around seven times the time taken by UDDI to publish a service, under the assumption that no additional ontologies have to be loaded to the registry. On the other hand, the time to process a query is in the order of milliseconds. While the above increases the time spent for publishing service advertisements, it considerably reduces the time spent to answer a user request compared to approaches based on on-line reasoning (e.g., see Figure 2). Indeed, the querying phase is reduced to performing lookups in the hierarchical data structure that represents the classified ontology, and to performing intersections between the lists that store information about the service advertisements. Thus, no on-line reasoning is required to answer a user request. However, the publishing phase still requires semantic reasoning on service descriptions which is an expensive process in terms of consumed resources.

On the other hand, solutions to reduce the number of matches performed to answer a user request are generally based on service classification. OWL-S specification provides the mean of defining hierarchies of service descriptions called *profile hierarchies*. These hierarchies are similar to the object-oriented inheritance hierarchies. For instance, when a new service profile is defined it may be specified as a subclass of an existing profile class. This allows the new service to inherit all the properties of all the classes specified in its super-hierarchy of classes. While this approach allows the classification of service profiles according to the classes from which they inherit, it does not allow considering possible relationships between service profiles that do not have the same common set of properties but that still provide similar functional features. Service classification can also be based on the service category using existing taxonomies such as NAICS[13] or UNSPSC[14]. However, service categories alone does not give enough information about the service functionality.

[13] http://www.census.gov/epcd/www/naics.html
[14] http://www.unspsc.org/

Using the matching relation defined in the previous section, we propose an efficient semantic service discovery protocol for pervasive computing environments. Efficiency is addressed in terms of response time for both the discovery and advertisement of service capabilities. Towards this goal, we present below a number of optimizations of the semantic matching process. First, in order to reduce the time to load and classify ontologies, which is the most costly phase in the discovery process, we propose to encode classified ontologies (§ 3.2). Then, in order to reduce the number of semantic matches performed in the querying phase we propose to classify capabilities of networked services into hierarchies (§ 3.3).

3.2 Encoding Concept Hierarchies

In order to avoid semantic reasoning at runtime we propose to encode classified ontologies, represented by hierarchies of concepts, using intervals as described in [3]. These hierarchies represent the subsumption relationships between all the concepts in the ontologies used in the directory. The main idea of the encoding is that any concept in a classified ontology is associated with an interval. These intervals can be contained in other intervals but are never overlapping. The intervals are defined using a linear inverse exponential function $linKinvexpP(x) = \frac{1}{p^{int(\frac{x}{k})}} + (x \bmod k) * \frac{1}{k} * \frac{1}{p^{int(\frac{x}{k})}}$, where p and k are two parameters to be fixed. Regarding the scalability of this encoding solution, experiments show that for p=2 and k=5, and a system encoding real numbers as 64 bits doubles, the maximum number of entries that we can have on the first level of the hierarchy is 1071 and the maximum number of levels that we can have on the first entries of a level is 462 levels. Figure 3 taken from [3], shows an example of encoding a hierarchy of concepts with intervals.

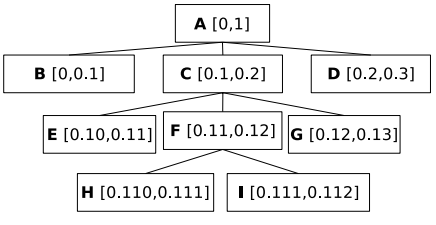

Fig. 3. Example of encoding a class hierarchy

Under the assumption that the classified ontologies are encoded and that service advertisements and service requests already contain the codes corresponding to the concepts that they involve, semantic service reasoning reduces to a numeric comparison of codes. Indeed, to infer whether a concept C1 represented by the interval I1 subsumes another concept C2 represented by the interval I2, it is sufficient to compare whether I1 is included in I2. In order to ensure consistency of codes along with the dynamics and evolution of ontologies, service advertisements and service requests specify the version of the codes being used. We assume that services periodically check the version of codes that they are using and update their codes in the case of ontology evolution.

3.3 Semantic Service Advertisement and Matching

Based on the encoding technique defined in the previous section, we present an algorithm for matching a requested capability with a set of capabilities of networked services. Service capabilities could be added or deleted at any time from the existing set of capabilities. When a request comes, the algorithm tries to find a capability that best matches the request minimizing the number of semantic matches performed with capabilities of networked services. At a pre-processing phase the algorithm classifies capabilities of networked services and constructs directed acyclic graphs (DAGs) of related capabilities. These graphs are indexed according to the ontologies being used in the capabilities that they contain. The relationship between capabilities that we consider to construct a graph is given by the relation $Match$ and the function $SemanticDistance$ defined in Section 2.3. Specifically, if both $Match(C_1, C_2)$ and $Match(C_2, C_1)$ hold and $SemanticDistance(C_1, C_2) = SemanticDistance(C_2, C_1) = 0$, then C_1 and C_2 will be represented by a single vertex in the graph. For all the other cases where $Match(C_1, C_2)$ holds, C_1 and C_2 will be represented in the graph by two distinct vertices with a directed edge from C_1 to C_2.

When a new service comes in the network the set of capabilities that it provides are classified among the existing hierarchies. The algorithm of classifying new capabilities in the existing hierarchies is described below.

When a request Req arrives, the algorithm first selects among the existing DAGs, graphs that contain services that are more likely to match the request. This is done using the indexes given to each graph, which correspond to the set of ontologies used by the capabilities of that graph. When a graph G is selected the algorithm performs a matching between the request and the most *generic* capabilities of this graph. These capabilities are said to be more generic than other capabilities contained in their subhierarchy because they provide a number of outputs that is greater or equal to the number of outputs of the other capabilities, and further because their provided outputs subsume the outputs of other capabilities (e.g., in Figure 1, the capability SendDigitalStream is *more generic* than the capability ProvideGame). These capabilities correspond to the capabilities represented by vertices of this graph that do not have predecessors, i.e., the set $Roots(G)$. Similarly, we define $Leaves(G)$ as the set of vertices in the graph G that do not have successors. If $Match$ between Req and all the capabilities of $Roots(G)$ does not hold, the group G is filtered out, and another group is selected. Nevertheless, if the matching between the request and a capability C of $Roots(G)$ holds, i.e., $Match(C, Req)$ holds, we evaluate the semantic distance between C and Req. If the distance is equal to zero, C is selected, otherwise the algorithm tries to find a capability C' from the successors of C such that $SemanticDistance(C', Req) = Min(SemanticDistance(C_i, Req))$, where C_i is a successor of C. The algorithm for answering a user request is presented in more details later in this section.

Adding a New Service Advertisement. At a pre-processing phase, a set of DAG graphs are constructed and maintained. Each time a new service advertisement comes in the network, the graphs have to be updated with the set of capabilities provided by the new service. The algorithm of classifying the capabilities of a new service within a

set of Graphs G_1, G_2,..., G_n is given below. For each capability C_i provided by the new service, the algorithm tries to find a graph G_i in which this capability will be integrated (Steps (1), (2)). A subset of graphs is preselected according to the ontologies being used by C_i. The algorithm first checks whether C_i can be inserted in the sub-hierarchy of one of the root nodes of G. This is done by verifying if there exists a node $Root_i$ in $Roots(G_i)$ such that $Match(Root_i, C_i)$ holds (step (3)). If $Match(Root_i, C_i)$ holds (step 8), then C_i will have a predecessor in G_i. The next step is to find this node, N_i, among the successors of the node $Root_i$, such that the $Match(Succ(N_i), C_i)$ fails, and to draw an edge from C_i to N_i. Moreover, C_i could have a successor in G_i. Thus, the algorithm tries to find among the set $Leaves(G_i)$ if there is a node $Leaf_i$ such that $Match(C_i, Leaf_i)$(step (9)). If $Match(C_i, Leaf_i)$ holds, then C_i will have a successor in G_i. The next step is to find this node, N_i, among the predecessors of $Leaf_i$ such that $Match(C_i, Pred(N_i))$ fails, and to draw an edge from C_i to N_i (step (11)). On the other hand, if $Match(Root_i, C_i)$ does not hold (step(4)), C_i will not have a predecessor in G_i. Nevertheless, C_i could have a successor in G_i. Thus, the algorithms checks whether there is a node $Leaf_i$ in $Leaves(G_i)$ such that $Match(C_i, Leaf_i)$ holds (steps (5), (6) and (7)). These steps are similar to the aforementioned steps (9), (10) and (11).

input: $C_1, C_2, ..., C_n$ the set of capabilities of the new service,
$G_1, G_2, ..., G_m$ the set of existing graphs.
output: $G'_1, G'_2, ..., G'_k$ the set of graphs after the insertion of the new capabilities.

InsertCapabilities(capabilities)

Forall the capabilities C_i in $C_1, ..., C_n$ **do**{ (1)
  For all the graphs G_i in $G_1, ..., G_m$ that use the same ontologies as C_i
  until the insertion of C_i **do**{ (2)
    For ($Root_i$ in $Roots(G_i)$) **do**{ (3)
      If ($\neg Match(Root_i, C_i)$) **then**{ (4)
        For ($Leaf_i$ in $Leaves(G_i)$) **do**{ (5)
          If ($\neg Match(C_i, Leaf_i)$) **then** (6)
            Fail;
          Else{ (7)
            Test with Predecessors of $Leaf_i$
            until $\neg Match(C_i, Pred_j(Leaf_i))$
            Draw an edge from C_i to $Pred_{j+1}(Leaf_i)$
          }}
      }**Else**{ (8)
        Test with Successors of $Root_i$
        until $\neg Match(Succ_j(Root_i), C_i)$
        Draw an edge from $Succ_{j-1}(Root_i)$ to C_i
        For ($Leaf_i$ in $Leaves(G_i)$) **do**{ (9)
          If ($\neg Match(C_i, Leaf_i)$) **then** (10)
            Fail;
          Else{ (11)
            Test with Predecessors of $Leaf_i$
            until $\neg Match(C_i, Pred_j(Leaf_i))$
            Draw an edge from C_i to $Pred_{j+1}(Leaf_i)$
          }}}}}}

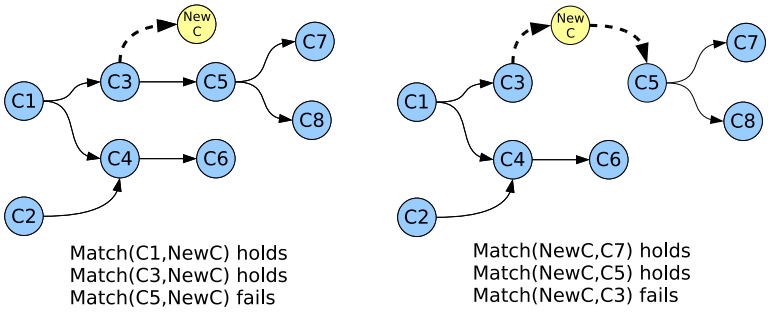

Fig. 4. Example of inserting a capability in a DAG

Figure 4 shows an example of inserting a capability, $newC$, in a DAG of capabilities, G. The first step (left part of the figure) is to match $newC$ with capabilities from $Roots(G)$ to find out whether $newC$ will have a predecessor in G. Indeed, $Match(C_1, new_C)$ holds, which means that one of the successors of C_1 will be linked with $newC$, i.e., C_3. The next step (right part of the figure) is then to find out whether $newC$ will have a successor in G. This is done by matching the capabilities in $Leaves(G)$ with $newC$. Indeed, $Match(newC, C_7)$ holds, which means that $newC$ will be linked with one of the predecessors of C_7, i.e., C_5.

Answering User Requests. When a user request that contains a set of required capabilities comes, the algorithm below finds out a set of capabilities of networked services that best match the ones required by the user. More precisely, for each capability C_i in the user request the algorithm tries to find a graph that may contain capabilities that match C_i (steps (1) and (2)). A graph G_i is selected if it is indexed with the ontologies used in the request and if there exist a node $Root_i$ in the set $Roots(G_i)$ such that $Match(Root_i, C_i)$ holds (step 3). In this case, a node that has the minimal semantic distance with C_i is selected from the successors of $Root_i$ (step 5).

inputs: a set of capabilities required in the service description $C_1, C_2, ..., C_n$
a set of graphs $G_1, G_2, ..., G_m$
outputs: a set of capabilities of networked services that match the capabilities given as input

MatchService(requested service)

For all the capabilities C_i required in the service description **do**{ (1)
  For all the graphs G_i in $G_1, ..., G_m$ that use the same ontologies as C_i
  until C_i is matched **do**{ (2)
    For all $Root_j$ in $Roots(G_i)$ **do** { (3)
      If ($\neg Match(Root_i, C_i)$) **then** (4)
        Try with the next node in $Root(G_i)$
      Else (5)
        Return $Succ(Root_i)$ from the successors of $Root_i$ such that
        $SemanticDistance(Succ(Root_i), C_i)$ is minimal
  }}}

An example of matching a requested capability with capabilities of networked services is given in Figure 5. In this figure, the requested capability $NewC$ uses the ontology

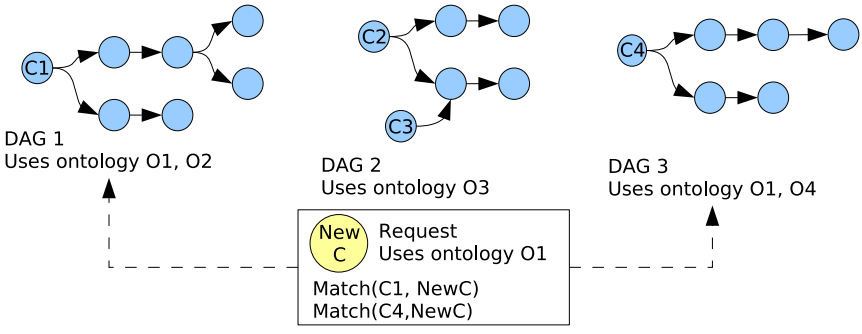

Fig. 5. Example of matching a user's requested capability

O_1 in its specification. This allows to filter out the DAG_2 as it is indexed with only the ontology O_3. The next step is to match $NewC$ with capabilities from $Roots(DAG_1)$ and $Roots(DAG_3)$, i.e., the capabilities C_1 and C_4. If the matching fails with one of these capabilities, we can infer that no capability will match $newC$ in the corresponding graph.

The benefits of using this solution to match user's required capabilities with capabilities of networked services is to reduce the number of semantic matches performed to answer a query. Indeed, it is sufficient to perform a semantic match with a subset of the capabilities of networked services rather than performing a semantic match with all the capabilities hosted by a directory of services. Furthermore, using the encoding of classified ontologies allows to reduce the semantic reasoning to a numeric comparison of codes.

4 S-ARIADNE Service Discovery Protocol

Towards the deployment of our solution in pervasive computing environments, we build upon the Ariadne middleware[15], which introduces a semi-distributed service discovery protocol for mobile ad hoc networks (MANETs) [12]. According to the design presented in [12], our discovery protocol, which we call S-Ariadne, relies on a backbone of directories constituting a *virtual network*. Directories are dynamically deployed, each directory performing service discovery in its vicinity. Then, service discovery in the global network is based on collaboration among deployed directories.

More precisely, S-Ariadne decomposes into a local and a global discovery process. The local discovery process is performed by each directory. Each directory is then responsible for:

(i) caching the Amigo-S descriptions of the services available in its vicinity, and classifying the capabilities provided by these services according to the grouping scheme discussed in Section 3.3, and
(ii) periodically advertising the presence of registered services to the vicinity.

[15] http://www-rocq.inria.fr/arles/download/ariadne/

When a directory receives a service request, specified as an Amigo-S service, it seeks capabilities of the cached services that semantically match the requested service as discussed in section 3.3.

To deal with the dynamics of pervasive networks, directories are dynamically and homogeneously deployed in the network using an on the fly election process. Specifically, if for a given period of time, a node does not receive any directory advertisement, the node initiates the election of a directory. The election process is done by broadcasting an election message in the network up to a given number of hops. Then, nodes can either accept or refuse to act as a directory, depending on a number of parameters such as network coverage, mobility and remaining/available resources. This mechanism allows electing directories with the best physical properties and distributing them efficiently since an election process is launched in the less covered areas. A node acting as a directory then periodically advertises its presence in its vicinity (i.e., up to a given number of hops).

The global service discovery process is based on collaboration among elected directories. However, the efficiency of the discovery process in terms of response time and generated traffic requires to query directories that are the most likely to cache service advertisements that do match the requested service. Towards this goal, we use directory categorization as introduced in [12], which gives a compact overview of the directory content. More precisely, we use Bloom filters for summarizing the content of a directory. The main idea is to compute a vector v of m bits, which corresponds to a Bloom filter. For any capability C, its semantic description relies on a set of ontologies $O(C) = \{O_1, O_2, ..., O_n\}$ to which belong the concepts describing its inputs, outputs and properties. Then, for each capability C provided by a networked service, and stored in a directory, the capability description in terms of used ontologies is hashed with k independent hash functions. Each ontology is considered in terms of its URI. The bits of the vector v whose positions are given by the results of the k hash functions are set to 1, i.e., the bits at position $h_1(O(C)), h_2(O(C)), ..., h_k(O(C))$ are set to 1. In order to determine whether a directory possibly caches a requested capability Req using the directory's Bloom filter, we check whether the bit positions $h_1(O(Req)), h_2(O(Req)), ..., h_k(O(Req))$ in the vector are all set to 1. If there is a bit that is not set to 1, the directory will not contain the required capability. Nevertheless, if all the bits are set to 1, the directory is likely to contain the required capability, and a concrete local service discovery is performed in that directory. The probability of a *false positive* depends on the parameters k that is the number of hash functions and m that is the size of the Bloom filter. These values can be chosen so that the probability of false positive is minimized.

The cooperation between directories is performed by exchanging the Bloom filters that give an overview of the directories content. The exchange of Bloom filters is done when new directories are elected and reactively, i.e., requested by another directory, when the percentage of false positives reaches a given threshold.

According to the deployment policy, each mobile node is associated to at least one directory. When the mobile node seeks a service characterized by a set of required capabilities, it sends a query message to the directory that is responsible of its network area (i.e., in its vicinity). The directory performs for each required capability a local service

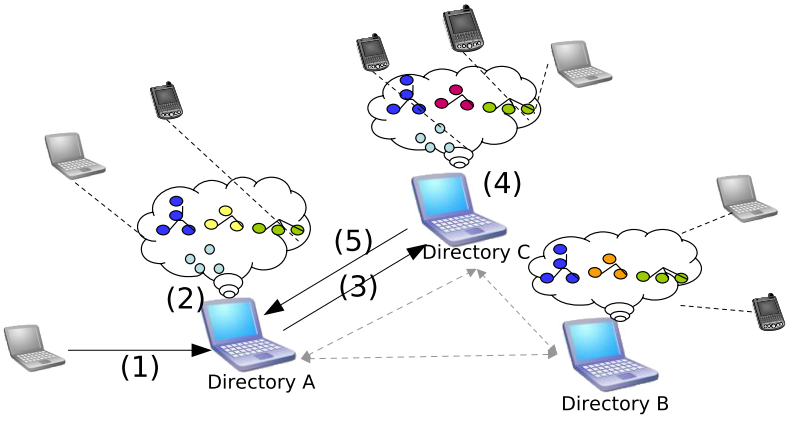

Fig. 6. S-Ariadne

discovery, as described in Section 3.3. If the required capabilities are not stored locally, the directory forwards the request to a subset of directories that are likely to cache capabilities that match the request. The directories to which the request is forwarded are selected according to their Bloom filters and additional parameters such as remaining battery lifetime and the distance between the respective directories.

Figure 6 provides an overview of the S-Ariadne architecture. In the figure, three nodes have been elected to act as directories. When a service request is issued, the directory node that is in the vicinity of the service requester, i.e., Directory A, receives the service request (Step(1)). The directory performs a local service discovery to find capabilities that semantically match the capabilities of the requested service (Step (2)). Service advertisements providing these capabilities are returned to the requester. If some capabilities have not been found locally, another request is sent to remote directories that are likely to store relevant capabilities according to their summarized description, i.e., Boom filters (Step (3)). These directories perform a local service discovery (Step (4)), and return the corresponding service advertisements (Step (5)), which are sent to the requester (Step(6)).

5 Prototype Implementation and Evaluation

We have implemented a prototype of our solution to efficient matching of semantic service capabilities as part of the Ariadne service discovery protocol extending it to S-Ariadne. We have evaluated the impact of introducing semantic service matching in Ariadne, which originally uses basic WSDL-based syntactic matching of Web services for the local service discovery. We have performed our evaluations on a Toshiba Satellite notebook with a 1.6 GHz Intel Centrino processor and 512 MB of RAM. In all the experiments that we performed, we increased the number of services from 1 to 100. The service descriptions are using 22 different ontologies, and each service description contains a single provided capability. Figure 7 shows the results of our first experiment, which evaluates the time to create graphs of services in an empty directory. A scenario

for this experiment would be realized when a directory leaves the network and when another one is elected and has to host the set of service descriptions available in its vicinity. Figure 7 shows three measurements: (1) the time to parse the service descriptions; (2) the time to classify the service capabilities into graphs; and (3) the total time, i.e., time to parse and create the graphs. From this figure, we can notice that the time to create the graphs is negligible compared to the time to parse service descriptions, i.e., XML parsing time, which is mandatory due to the use of Web services and Semantic Web technologies.

The results given by the second experiment that we performed are depicted in Figure 8. This experiment shows the time to insert a new capability in a directory. This figure shows 3 measurements: (1) the time to parse the new service description; (2) the time to insert a capability in a directory; and (3) the total time, i.e., the time to parse and insert the new service description. Results show that the to time to classify a capability in a set of existing graphs is negligible compared to XML parsing time of the service description. We also notice that this time is nearly constant. This is due to the fact that the number of semantic matches performed in the directory in order to insert a capability depends neither on the total number of services on the directory nor on the number of graphs. The time to insert a capability depends on the number of capabilities contained in the graph in which the capability will be inserted. This is due to the fact that graphs are indexed using the ontologies that are being used in the capabilities' descriptions, which allows pre-selecting a subset of graphs that are likely to be appropriate for the insertion of the new capability. Thus, only a few number of semantic matches are performed in order to insert a capability in a directory. The results of the third experiment

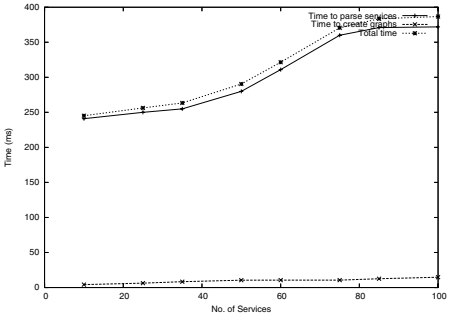

Fig. 7. Time to create graphs

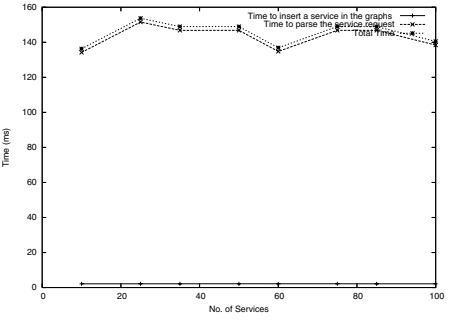

Fig. 8. Time to publish a service advertisement

that we performed are depicted in Figure 9. In this experiment, we evaluate the time to match a service request with services hosted by a directory. Furthermore, we compare the time to match a request in a directory where capabilities are classified into a set of graphs, with the time to match a request in a directory without classification. Results are given without the XML parsing time of the request description. In this figure, we can notice that without classification the average overhead for matching is around 50% of the time to match when the capabilities are pre-classified. Moreover, we can notice that

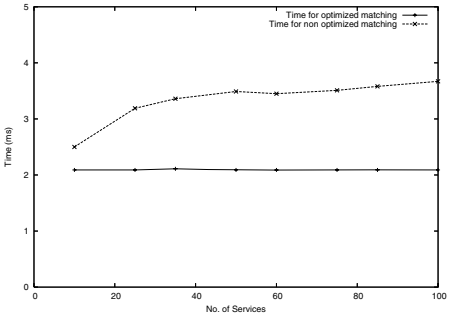

Fig. 9. Time to match a service request

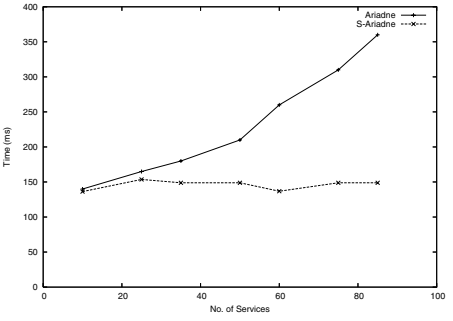

Fig. 10. Ariadne vs S-Ariadne

the time to match a request in the classified directory is almost constant, which is due to the graphs indexing and the directory structuring. We can also notice that the response time to match a required capability, excluding XML parsing time, is in the order of few milliseconds.

The last experiment that we performed is a comparison of the response time given by the classical syntactic-based matching performed by Ariadne and the optimized semantic matching performed by S-Ariadne. The results are given in Figure 10. This figure shows that the response time given by Ariadne is increasing with the number of services available in the directory, while S-Ariadne has an almost stable response time, which is due to the following reasons: (1) using S-Ariadne, the services are parsed once at the publishing phase and their capabilities are classified, which avoids matching a request with all the services of the directory; (2) due to the numeric encoding of classified ontologies, the semantic matching performed by S-Ariadne reduces to a numeric comparison of codes, while using Ariadne the matching is performed by syntactically comparing the WSDL descriptions. We can conclude that, using S-Ariadne, semantic matching, which allows to leverage the openness of pervasive computing environments, can be performed more efficiently than classical syntactic matching. Furthermore, thanks to the indexing and classification of service capabilities, S-Ariadne is more scalable than Ariadne.

6 Conclusion

The pervasive computing vision implies that everywhere around us the environment is populated with networked software and hardware resources that can be discovered and integrated towards the realization of our daily tasks. Towards the realization of this vision, middleware support for the efficient dynamic discovery of software and hardware resources of the pervasive computing environment is a key requirement. Such middleware support has to deal with the heterogeneity of the networked resources. This can be partially addressed using service-oriented architectures, and particularly the Web services paradigm. Indeed, Web services enable having a homogeneous vision and access to the heterogeneous networked resources of the environments. Nevertheless, Web

services discovery and interaction commonly relies on the syntactic conformance of service interfaces, for which common understanding is hardly achievable in open environments. The Semantic Web paradigm allows to overcome this limitation by introducing semantic specification of service functional and non-functional features, which enables semantic reasoning on Web services capabilities.

Building on semantic Web services, our approach to dynamic service discovery in pervasive computing environments relies on the Amigo-S language for the semantic specification of pervasive services, and introduces an efficient matching relation of service capabilities, which we have integrated in S-Ariadne extending Ariadne, a semi-distributed discovery protocol adapted to pervasive computing environments. Our solution optimizes the costly ontology-based semantic reasoning on one hand, and the number of semantic matches to be performed to answer a user request on the other hand. The optimization of the semantic reasoning is based on the encoding of classified concept hierarchies, which allows to reduce the semantic reasoning to a numeric comparison of codes, while the optimization of the matching process is based on the classification of service capabilities into hierarchies of related capabilities. Our results show that S-Ariadne provide better response time for the semantic matching of service capabilities than Ariadne, its syntactic ancestor, for the basic syntactic service matching. Furthermore, thanks to the indexing and the structuring of service directories, S-Ariadne is more scalable than a classical service directory.

Acknowledgments

This research is partially supported by the European IST AMIGO project[16] (EU-IST-004182).

References

1. Yerom-David Bromberg and Valerie Issarny. Indiss: Interoperable discovery system for networked services. In *Proceedings of ACM/IFIP/USENIX 6th International Middleware Conference (Middleware'05)*, 2005.
2. Amigo Consortium. Amigo middleware core: Prototype implementation and documentation. Project Deliverable D3.2, 2006.
3. Ion Constantinescu and Boi Faltings. Efficient matchmaking and directory services. In *Proceedings of the IEEE International Conference on Web Intelligence (WI'03)*, 2003.
4. N. Georgantas, S. Ben Mokhtar, Y.-D. Bromberg, V. Issarny, J. Kalaoja, A. Grodolle J. Kantarovitch, and R. Mevissen. The amigo service architecture for the open networked home environment. In *Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, 2005.
5. Nikolaos Georgantas, Sonia Ben Mokhtar, Ferda Tartanoglu, and Valerie Issarny. *Architecting Dependable Systems III*, chapter Semantic-aware Services for the Mobile Computing Environment. Springer Verlag, 2005.
6. Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference of Very Large Data Bases, VLDB'95*, 1995.

[16] http://www.hitech-projects.com/euprojects/amigo/

7. Valerie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Francoise Sailhan, Rafik Chibout, Nicole Levy, and Angel Talamona. Developing ambient intelligence systems: A solution based on web services. *Journal of Automated Software Engineering*, 2004.
8. Sonia Ben Mokhtar, Damien Fournier, Nikolaos Georgantas, and Valerie Issarny. Context-aware service composition in pervasive computing environments. In *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques (RISE'05)*, 2005.
9. Sonia Ben Mokhtar, Anupam Kaul, Nikolaos Georgantas, and Valerie Issarny. Towards efficient matching of semantic web service capabilities. In *Proceedings of the workshop of Web Services Modeling and Testing (WS-MATE'06)*, 2006.
10. Sonia Ben Mokhtar, Jinshan Liu, Nikolaos Georgantas, and Valerie Issarny. Qos-aware dynamic service composition in ambient intelligence environments. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, 2005.
11. Mike P. Papazoglou. Service -oriented computing: Concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering(WISE '03)*, 2003.
12. Francoise Sailhan and Valerie Issarny. Scalable service discovery for MANET. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom'05)*, 2005.
13. Naveen Srinivasan, Massimo Paolucci, and Katia Sycara. Adding owl-s to uddi, implementation and throughput. In *Proceedings of the Workshop on Semantic Web Service and Web Process Composition*, 2004.